Getting Started in R

Part 2

by M. Papathomas and E. Rexstad (with material from a document prepared by R. King)

```
## Warning: package 'knitr' was built under R version 3.2.5
```

Solutions to questions

In the last Microlab session, we introduced the computer package R and investigated its basic operating commands. Usually, we need to perform more complex operations and/or repeat the same operations a number of times. In these cases it is useful to write our own functions within R. We will pay attention to writing such functions and understanding written R codes within this practical session, building on the basic commands from the previous Microlab handout *Getting Started in R*. Refer back to the previous handout if necessary.

We initially consider how to read in a datafile in R before considering some useful existing functions in R.

Reading in Data in R

Data files for R are usually in the form of a text file. They usually have a structure like in the example below.

```
ID  Gender  Height_cm Weight_kg
1    M       170       78
2    M       182       86
3    F       168       72
4    F       175       76
```

Depending on who created the data file, it may appear more or less messy, with the columns not aligned properly, or the Header with the names of the variables missing. R can still read the file accurately, but you should always check that the end result is correct.

If you need to read in data from a data file you should either create the data file (using for example Notepad) and save it on the H: drive, or if it already exists, you should copy it to the H: drive. You may want to create a sub-directory on your H: drive just for R files. You will find two example data sets on MMS, one named data_1.txt (the file above) and another named data_2.dat. The extension of the file is not important when R reads data in from it. Assuming that you have copied those files to your H: drive, you can read data_1.txt with the following command, changing the path accordingly. Note the forward slashes in the directory path name.

```
dfdata_1 <- read.table("H:/myname/myRfolder/myRdata/data_1.txt",header=TRUE,sep="")
```

To find out more about this function use ?read.table. This command gives the location of the file, plus the information that the first line of the file contains the names of the variables, and the names and observations are separated with one or more spaces. It reads the data set into a data frame called dfdata_1. A data frame is a collection of variables. To work with one of the variables you need to use the name of the data frame, combined with the name of the variable after the $ symbol. For example, if you want to add the four heights,

```
sumOfHeights <- sum(dfdata_1$Height_cm)
sumOfHeights
```

```
## [1] 695
```

To save typing the pathname to the data file repeatedly, we can change what is called the `working directory`. This way, every time we read a file of data, `R` will already know where to find it and all we need to do is to tell `R` what the file is called. In addition, if you wish to save any graphs plotted in `R` for example, `R` will automatically suggest saving the plot in the given directory on your `H:` drive. In R-Studio, use the `File` tab of the lower right panel. Use the $\cdots$ button on the far right of the panel to navigate to the directory of interest (perhaps `H:` in the MicroLab), then use the `More` button in that panel and select `Set as working directory`. This can also be done using the `setwd()` command in `R`.

Another way to read a data file into `R`, is to use `R studio` and click on the `Import Dataset` icon on the top right window. The option `From Text File` will take you to the File Manager. When you find your file you will be presented with a dialogue window, where you can change the name of the data frame, indicate if there is a Header in the file, etc.

Try these on your own

- Download the file `data_2.dat` from MMS.
- Read the data into a data frame called `dfdata` in `R` (change the working directory of `R` to where the file was downloaded) using the `read.table()` function. `data_2.dat` is a messy file that contains observations arranged in 30 rows and 10 columns.
- Find the dimension of `dfdata`. (Hint: the function `dim()` may be useful).
- Calculate the mean of each column of `dfdata` using the `colMeans()` function.
- Calculate the mean of each row of `dfdata`.
- Find the maximum and minimum values of `dfdata`.
- Convert the data frame to a standard array using `data2 <- as.matrix(dfdata)` and plot a histogram of the data. (What happens if you do not convert the data frame to an array?).
- Plot a histogram of the 1st row and 1st column. (Hint: for an array named `dummy`, `dummy[i,]` and `dummy[,i]` correspond to all elements of the ith row and column, respectively).

Solution

Data are read into an object (in fact a data frame) called `dfdata` using the `read.table()` function **after** setting the working directory (`setwd()`) to the directory where I saved the file after downloading from MMS. Note the forward slashes in the directory path name.

```
setwd("C:/Users/XPS/OneDrive/Documents/teaching/R intro material/docs/2015_2016")
dfdata <- read.table(file="data_2.dat")
```

To double check what type of object was created by `read.table()` use the `class()` function.

Result of the `dim()` function applied to the object `dfdata` produces:

```
dim(dfdata)
```

```
## [1] 30 10
```

Functions exist to calculate the mean of each row (or column) e.g.

```
rowMeans(dfdata)      # colMeans(dfdata) would provide column means
```

```
##  [1]  0.379057692 -0.546325130  0.142534799 -0.023754014  0.430772809
##  [6]  0.201492696  0.190566089 -0.381813039  0.392542922  0.024511019
## [11]  0.003848438  0.240078255  0.267697511  0.046038869  0.071305288
## [16]  0.008342353 -0.368127757  0.161686397 -0.267657660  0.497611455
## [21] -0.509909470  0.087311933 -0.230139041  0.098109633  0.019709365
## [26]  0.121995269 -0.085000766  0.051986275 -0.643776056  0.041594116
```

Functions `min()` and `max()` can operate across both dimensions of our two-dimenstional object `data`

```
min(dfdata)
```

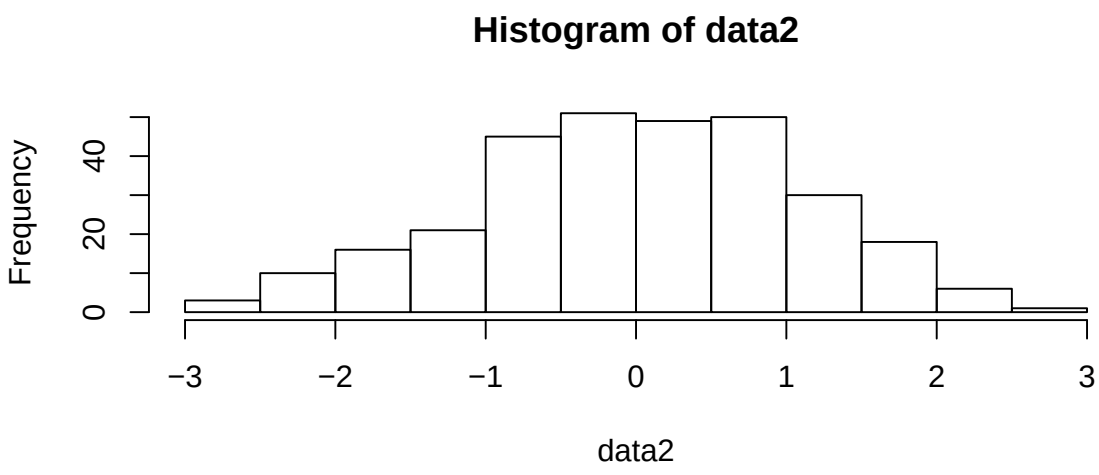
```
## [1] -2.879658
```

```
max(dfdata)
```

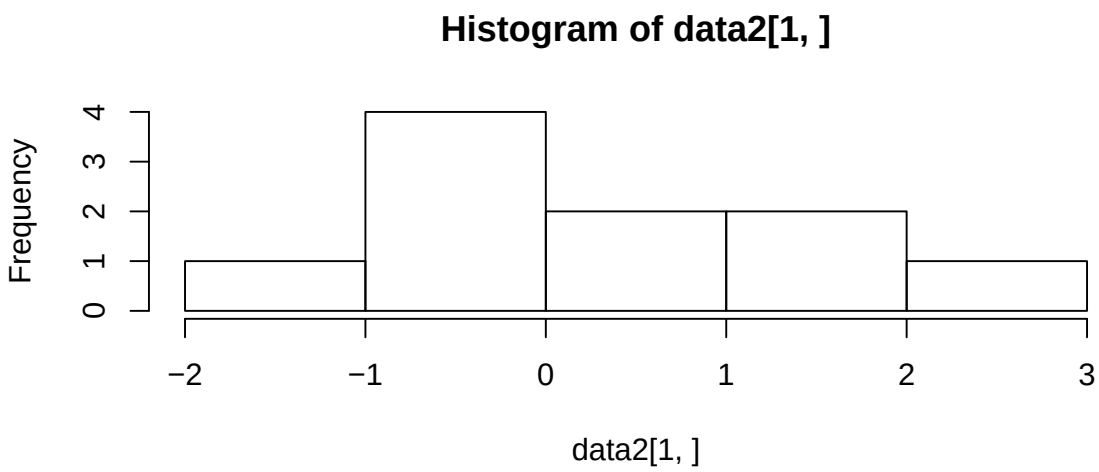
```
## [1] 2.700584
```

To create a histogram for the observations and for the first row only,

```
data2 <- as.matrix(dfdata)
hist(data2)
```

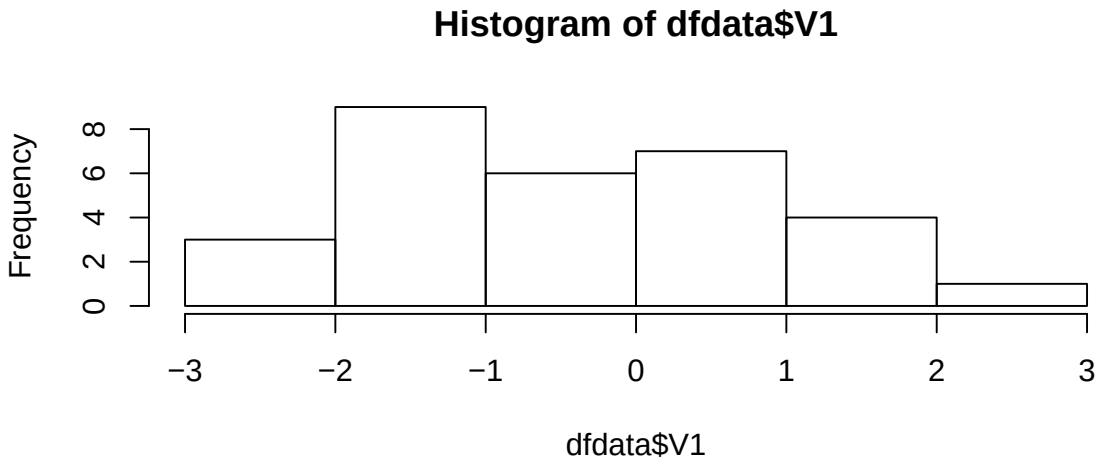



```
hist(data2[1,])
```



We will create a histogram for the first column of data in a different way now. This column received a default name when read by `read.table()`, `V1` (for *variable* 1)

```
hist(dfdata$V1)
```



Advanced topic: if we wanted to create a histogram for each column of our data frame, we could use the fact that a data frame is a *list*. Elements of lists can be specified by using double brackets: `[[]]`. To reference the first column in a call to the `hist()` function, we could specify:

```
hist(dfdata[[1]])
```

Hence,

```
i <- 4
hist(dfdata[[i]])
```

would show a histogram for the fourth column of our data frame `data`.

Writing Functions in R

We now create a function in R, to simulate n tosses of a (potentially) biased coin, to record the number of tails recorded where there are m sets of n tosses. You can do this simply by entering the following commands in R. However, you should get into the habit of creating functions using the R-Studio editor panel. You can save them in your workspace for future use, and if you make an error, you can go back to the code in the editor, and correct it, rather than enter all the commands again (remember to change the working directory to the directory where you have created your file containing the function).

Into the R-Studio editor panel type:

```
coin.function <- function (n, p, m) {
  tails <- rbinom(m,n,p)
  results <- sum(tails)
  return(list(tails = tails, sumtails = results, n = n, p = p, m = m))
}
```

Highlight the text, and click the `Run` icon at the top of the Editor panel, or use the keyboard shortcut `<ctrl>-Enter`.

If you get an error message, return to the Editor pane, correct the error, and resend. This creates a function coin.function. If you do not know what any of the individual R commands do within the function `coin.function()`, use the `help()` function. The `return()` function sends an object out of the function. We place the function `list()` inside the call to `return()` to combine several objects into a single object to return to the calling code.

The function `coin.function()` has three arguments: `n`, `p` and `m`, corresponding to the number of binary trials, the probability of success in each, and the number of simulations of this binomial distribution, respectively. The last line returns the values we want, with names `tails`, `sumtails`, `n`, `p` and `m` to the code that called the function.

To run the function, simulating 5 random variables each from a Bin(100,0.5), type:

```
tosses <- coin.function(n=100,p=0.5,m=5)
tosses
```

It is not necessary to include the argument names when running the function. Equivalently, we can use `tosses <- coin.function(100,0.5,5)`.

Typing `tosses` will echo the contents of the object `tosses` to the console. If you were only interested in the total number of tails obtained in each experiment (i.e. the term `tails`) these can be outputted using `tosses$tails` (and similarly for the other listed output values).

Rerun the function. Note that you will obtain a different answer - this is an example of the replication principle.

Save the function in a file `function.R` for use in another R session at another time. The function can then be read into the R workspace using the function:

```
source("function.R")
```

(Remember to set the working directory as the directory containing the function - else specify the full directory within the `source()` function). Alternatively you can use the drop-down menu of R-Studio using `Code | Source File` or the keyboard shortcut `Ctrl+Alt+G`.

Try these on your own

- Consider the term `sumtails`. What is the distribution of this parameter?
- Modify `coin.function()` function so that the number of tails that are tossed in each experiment are plotted in a histogram where we have $n = 50$ tosses of the coin, the coin is weighted, so that $p = 0.05$ and we repeat the experiment $m = 5$ times.

  - What happens if we increase m?
  - Calculate the empirical 2.5% and 97.5% quantiles of the distribution (i.e. output the 2.5% and 97.5% quantiles of the values simulated - the function `quantile()` may be useful here).
  - How do these compare with the true values from the $Bin(100, 0.5)$ distribution (the `R` function `qbinom()` may be useful here)?

- Write a function to simulate 1000 random deviates from a $Po(\lambda)$ distribution. Set $\lambda = np$, for n and p given above, for the weighted coin. Compare the simulated values with those obtained from a $Bin(n, p)$ distribution (for example looking at quantiles and/or histograms). Comment on the results.

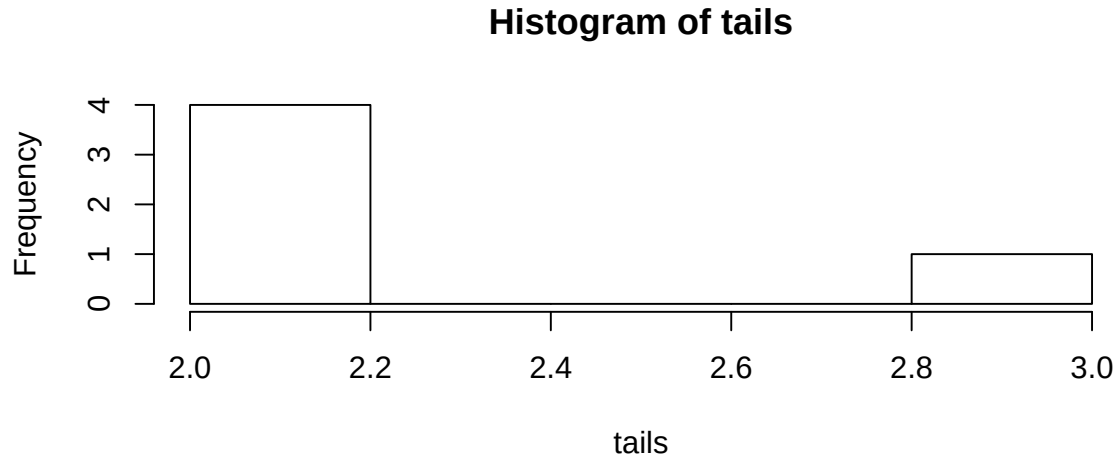
Solution

- The term `sumtails` sums the number of tails obtained for each of the `m` Binomial identical and independent experiments. Recall that the Binomial distribution is itself the sum of `n` Bernoulli trials with probability `p` of success. Thus `sumtails` is simply the sum of the number of `m` $\times$ `n` Bernoulli trials, independently, each with probability p of success. Thus `sumtails` has a $Bin(nm, p)$ distribution - in this case it is $Bin(500, 0.5)$.
- We could enhance our function by the addition of a histogram as well as incorporate the `quantile()` function by:

```
coin.function <- function (n, p, m) {
  tails <- rbinom(m,n,p)
  hist(tails)
  qtails <- quantile(tails,probs=c(0.025,0.975))
  return(qtails)
}
```

Calling my revised function, produces this result

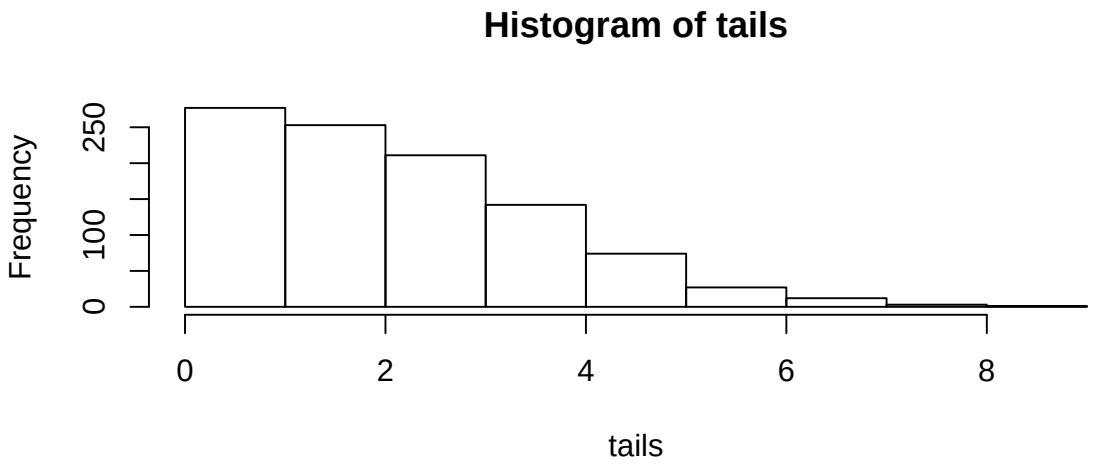
```
coin.function(n=50,p=0.05,m=5)
```




```
##  2.5% 97.5%
##   2.0   2.9
```

If I increase `m`, to increase the number of binomial simultions performed, the results are

```
coin.function(n=50,p=0.05,m=1000)
```




```
##  2.5% 97.5%
##     0     6
```

A heightened number of simulations produces results that are less influenced by stochastic fluctuations.

There is an `R` function that can compute the exact values of the $Bin(100, 0.05)$ distribution:

```
qbinom(c(0.025,0.975),50,0.05)
```

```
## [1] 0 6
```

The closeness of the `qbinom()` result to the result of our `coin.function()` result with a large number of replicates provides some empirical support for the estimation approach where we simulate from a distribution and use the simulated values to obtain summary statistics of the distribution is called *Monte Carlo integration*.

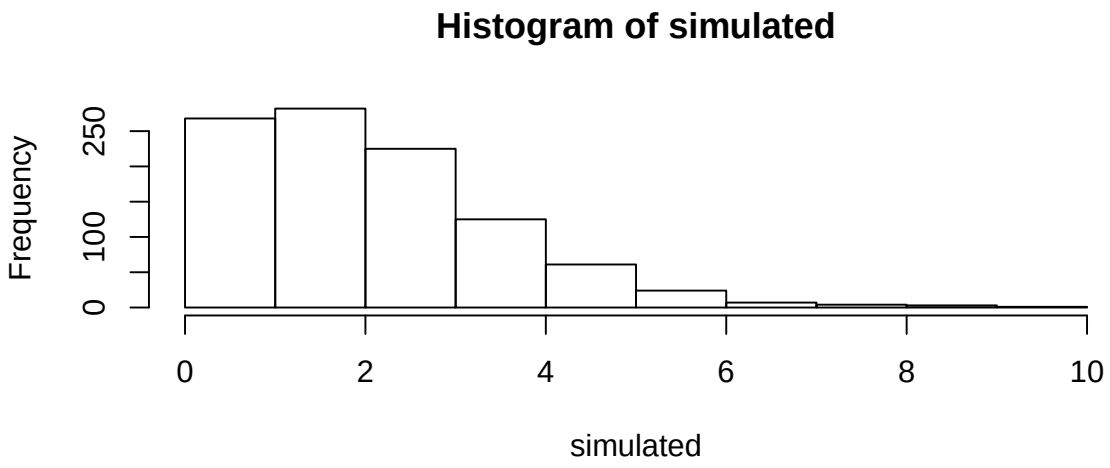
- A function to generate deviates from a Poisson distribution might be

```
po.function <- function (m, lambda) {
#    Function to create Poisson-distributed deviates, plot their distribution and return tails
  simulated <- rpois(m,lambda)
  hist(simulated)
  qpo <- quantile(simulated,probs=c(0.025,0.975))
  return(qpo)
}
```

Note, as you begin to create a pantheon of functions, it would be good practice to being adding some *documentation* to the functions to indicate what they do, why they were created, etc. That is the purpose of the brief comment line in `po.function()`.

Setting the arguments of `po.function()` to λ=2.5 and `m`=1000:

```
po.function(m=1000, lambda=2.5)
```




```
##  2.5% 97.5%
##     0     6
```

This result mimicks result of `coin.function()` because the $Bin(n,p)$ distribution is well-approximated by a $Po(np)$ distribution for $n \geq 20$ and $p \leq 0.05$ (to be more specific the Binomial distribution tends to the Poisson distribution as $n \to \infty$ and np remains fixed).

For Loops

When writing function in `R`, we often want to perform a given operation a number of times. A `for` loop allows us to repeat operations a given number of time. The general syntax for these loops are of the form

```
for (i in 1:n) {
                          # Add here the operation you want performed repeatedly
}
```

This means for $i = 1, \ldots, n$ perform the operation within the {} terms.

Example: Create a 6×6 matrix with all non-diagonal elements zero, and diagonal $(2, 4, 6, 8, 10, 12)$.

```
matr <- matrix(rep(0,36), nrow=6, ncol=6)
for (i in 1:6) {
  matr[i,i] <- 2*i
}
```

Note that a more efficient way of doing this would be

```
matr <- matrix(rep(0,36), nrow=6, ncol=6)
diag(matr) <- (1:6)*2
```

If Statements

The `if()` function in `R` allows us to perform operations only if a given condition is satisfied. The general syntax is of the form:

```
if (i == 1) {
                    # Add here operation to perform if i = 1
} else {
                    # Add here operation to perform if i not equal to 1.
}
```

Note that the `else` command is optional - we may not want to make any further operations if $i \neq 1$. Note also that `else` and the surrounding outward-facing braces exist on a line alone. Within the `if()` function we can use the following conditions:

Command	Meaning
==	"equal to"
!=	"not equal to"
>	"greater than"
>=	"greater than or equal to"
<	"less than"
<=	"less than or equal to"

Example: You pay 1 pound to play the following game. You throw one die, and if the outcome is 5 or greater you win 2 pounds, otherwise you win nothing. Write some code that simulates playing this game 100 times, and gives you the amount of money you are left with, assuming that you had 100 pounds when you started.

```
amount <- 100
for (i in 1:100) {
  dice <- sample(c(1,2,3,4,5,6),1)
  if (dice>=5) {
    amount <- amount+1
  } else {
    amount <- amount-1
  }
}
print(amount)
```

Try this on your own

- Change the code above to make the game more fair. Check if the game now seems to be fair by playing it many more times, so that you reduce the variability in the final outcome due to the simulation.

Solution

- Change the winning threshold to 4, and play the game 100000 times (or more), changing the initial amount accordingly. If the game is fair, the final amount should be very close to 100000. The more times you play, the closer the initial and final amounts should be.

```
amount <- 100000
for (i in 1:100000) {
  dice<-sample(c(1,2,3,4,5,6),1)
  if (dice>=4) {
    amount <- amount+1
  } else {
    amount <- amount-1
  }
}
print(amount)
```

```
## [1] 100186
```

Write Commands

A useful function, used when a function is not working properly, is the `print()` function; used to print object values to the console. Combining `print()` with the function `paste()` allows combining character strings with your object for labelling your results. For example, for the `coin.function()` function above add the lines:

```
print(paste("number of random variables =",m))
print(paste("and parameters of Binomial distribution =",n,p))
```

This sends the corresponding values to the console (i.e. the values `m` and `n, p`).

Example: Simulating Random Deviates

To check our calculations that gave us the following probability distribution for the $Bin(5, 0.4)$ distribution. (In practice, we are unlikely to need to check such a simple calculation, especially as we can obtain the true distribution from `R`, but it serves to illustrate how the method is applied in more complex circumstances.)

x	0	1	2	3	4	5
$f(x)$	0.0778	0.2592	0.3456	0.2304	0.0768	0.0102

The following R program generates 100 random deviates from this distribution, and outputs the proportion of deviates that equal 0,1,2,3,4,5:

```
nsim <- 100                      # nsim is number of random deviates to be generated
n <- 5                           # Read in n
p <- 0.4                         # Read in p
deviate <- rbinom(nsim,n,p)      # Simulate nsim random deviates from Bin(n,p) and
                                 # place them in the vector deviate
freq <- array(0,n+1)             # Create vector to record the frequency of each
                                 # possible value
for (i in 1:nsim){               # Loop over each random deviate, counting the
  j <- deviate[i]+1              # number of times each possible value is generated
  freq[j] <- freq[j]+1
}
freq <- freq/nsim                # Calculate the proportion of times each value is
                                 # simulated
print(freq)                      # Print out these proportions
```

Running the code a single time I obtained the output,

```
## [1] 0.10 0.28 0.35 0.25 0.01 0.01
```

Try these on your own

- How do the proportions I obtained agree with those in the above table? Why is this?
- Rerun the code for 10,000 simulations and output the observed frequencies. How do these compare the the values in the table above?
- Convert the `R` code above into a function using the R-Studio Editor panel, with `nsim`, `n` and `p` passed into the function as arguments and returning `freq`. Check that this function works by submitting the code to the console (`Run` icon) so the function is in the `R` workspace and re-running the simulations above.

Solution

- The proportions are "similar" to the probabilities in the table but there are discrepancies due to the replication principle. We are only obtaining a sample of values simulated from the distribution.
- Running the code for 10000 simulation produces this result:

```
## [1] 0.0789 0.2628 0.3502 0.2206 0.0771 0.0104
```

These results are quite close to the values presented in the table; the larger number of simulated values more accurately represent the Binomial probabilities.

- Converting the code to a function is a matter of converting the assigment statements in the first three lines to arguments to the function and incorporating a `return()` function around the final value computed by the function.

```
bin.function <- function(nsim,n,p) {
#     Create binomially-distributed deviates
  deviate <- rbinom(nsim,n,p)
  freq <- array(0,n+1)
  for (i in 1:nsim){
      j <- deviate[i]+1
      freq[j] <- freq[j]+1
  }
  freq <- freq/nsim
  return(freq)
}
```
